Investigation and Implementation of Matrix Permanent Algorithms for Identity Resolution

Marie-Odette St-Hilaire
Olivier Audet
Mathieu Lavallée

Prepared By: OODA Technologies Inc.
4891 Grosvenor
Montréal (Qc), H3W 2M2
514.476.4773

Prepared For: Defence Research & Development Canada, Atlantic Research Centre
9 Grove Street, PO Box 1012
Dartmouth, NS
B2Y 3Z7
902-426-3100

Scientific Authority: Dominic E. Schaub
Contract Number: W7707-145677
Call Up Number: 1;4501131255
Project: Investigation and Implementation of Matrix Permanent Algorithms for Identity Resolution
Report Delivery Date: March 31, 2014

Defence Research and Development Canada
Contract Report
DRDC-RDDC-2014-C291
December 2014

Identification of tracks/targets forms a cornerstone of Maritime Situational Awareness (MSA) in both tactical and operational settings. Resolving the identities of unknown targets often demands significant resources, and thus it is highly desirable to improve automatic vessel identification to the greatest extent that is computationally possible. Recent work at Defence Research and Development Canada (DRDC), Atlantic Research Centre has studied the statistical identification problem and found that it depends crucially on calculation of the permanent of a matrix whose dimension is a function of target count [21]. However, the optimal approach for computing the permanent is presently unclear.

The primary objective of this project was to determine the optimal computing strategy(-ies) for the matrix permanent in tactical and operational scenarios. As the exact calculation is #P-hard, approximation methods are necessary. However, many new and promising methods lack characterization, particularly in regard to their suitability for MSA applications. The present work seeks to clarify the computational options available and includes the following elements:

1. Identification and characterization of algorithms described in the scientific literature.
2. Implementation of those algorithms most suited for MSA.

This page is intentionally left blank.

Contents

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

List of symbols/abbreviations/acronyms/initialisms

CPU Central Processing Unit

DRDC Defence Research and Development Canada

FPRAS Fully Polynomial-Time Randomized Approximation

GPU Graphics Processing Unit

IS Importance Sampling

MCMC Markov Chain Monte Carlo

MSA Maritime Situational Awareness

RAM Random-Access Memory

ZDD Zero-suppressed Binary Decision Diagrams

This page is intentionally left blank.

Part 1

Scope

The computation of the permanent was identified as the main bottleneck of the statistical identification algorithm developed by Schaub [21]. This report presents efforts to identify and implement the best suited algorithms to calculate and approximate the permanent.

The context in which the permanent is computed, namely the identification of tracks/targets for Maritime Situational Awareness (MSA), imposes several requirements on the selected permanent algorithm performance and characteristics:

- Control on the error, i.e. the algorithm must produce stable results within a predictable error;
- Potential to run near real-time;
- Can process matrices of size up to 10,000;
- Non-negative square real matrices;
- No typical matrix characteristic, such as density, except the fact that it may have duplicated rows and/or columns;
- Will be run on a performing multi-core machine.

The report is divided as follow:

1. Section 2 presents an overview of algorithms summarizing pros and cons regarding the cited above requirements.
2. Section 3 presents the selected strategies for computing the permanent.
3. Section 4 presents the strategies selected for approximating the permanent.
4. Section 5 gives results, in terms of execution time and precision, for the selected computing algorithms. It also provides the reader with some recommendations and way aheads.

This page is intentionally left blank.

Part 2

Algorithm Overview

Several algorithms were developed to calculate and approximate the permanent of a matrix. The permanent of a matrix occurs while solving several different problems from various fields such as combinatorics, quantum field theory and identity resolution like in this current case.

Figure 2.1 provides an overview of the existing approaches to compute the permanent. There are mainly two strategies: the exact calculation and the approximation. The approximation algorithms can be divided into two groups: fixed and varying error approximations. There are considerably fewer algorithms with control on the error than without and they offer poorer performance in terms of scaling.

The following provides a brief review of algorithms developed into each category of Figure 2.1. It does not explore the details of the maths but rather points the most recent literature on the topic and their relevance for regarding our requirements.

For the remaining of this document, the permanent of a n by n matrix $A = (a_{ij})$ is noted $per(A)$. The notation $O(f(n))$ refers to the complexity of an algorithm, where $f(n)$ is the number of operations required by the algorithm which is a function of n.

2.1 Calculation Approaches

Only few algorithms exist to compute exactly the permanent of a matrix, namely the definition (that we refer to as the naive algorithm), Ryser algorithm (with or without the Gray code implementation), the Glynn algorithm [9] which has the same order of operation number than the Ryser algorithm and the Zero-suppressed Binary Decision Diagrams (ZDD).

The reason for this lack of options is the fact that the computation is known to be $\#P$ hard [25].

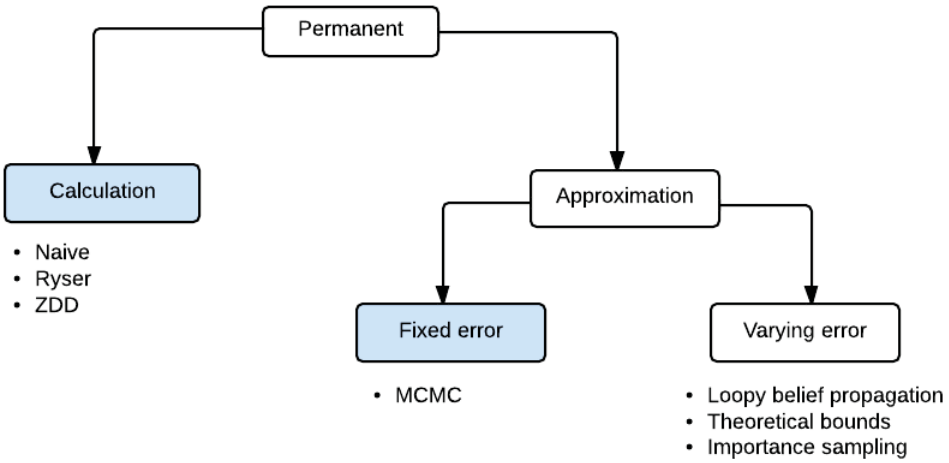

Figure 2.1: An overview of the most popular approaches to compute a permanent. The focus of this work are methods belonging to the blue boxes.

2.1.1 Naive

The naive algorithm is the implementation of the definition:

$$per(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}, \tag{2.1}$$

where S_n is the group off all $n!$ permutations of $N = \{1, 2, ..., n\}$. This formulae requires summing over all $n!$ permutations with N multiplications for each sum, resulting in a $O(n!n)$ algorithm.

2.1.2 Ryser

Ryser [20] came up with a different way to compute the permanent requiring less operations:

$$per(A) = (-1)^n \sum_{S \subseteq \{1,...,n\}} (-1)^{|S|} \prod_{i=1}^{n} a_{i,j}. \tag{2.2}$$

This algorithm is $O(n^2 2^n)$ down to $O(n2^n)$ by processing sets S in Gray code order. The implementation of this algorithm is described in section 3.

2.1.3 Zero-suppressed Binary Decision Diagrams

The ZDD is a tool used for solving various combinatorial problems (see [16] for details and applications to a wide variety of problems) and thus can be applied to compute a permanent[1]. The

[1]See sections 2.2.3 and 4 for more details on the combinatorics interpretation of the permanent

ZDD-based method is described in [8] for 0-1 and non-negative matrices. The authors found that for dense matrices, Ryser's formula requires less memory accesses, but for sparse matrices the ZDD-based method is more efficient. They also proposed a pruning strategy to sparcify dense matrices.

Although it is an interesting result, we found the algorithm not straightforward to implement, especially for dense matrices which require pruning. Moreover, the results presented did not convince us that the performance gain (over Ryser's algorithm) outweigh the cost of complexity.

2.1.4 Special cases

For certain types of matrices, the permanent turns out to be a function of the determinant. The determinant being computationally less expensive than the permanent[2], this property is very interesting. However, it is not clear how labour intensive it is to verify if a matrix, especially a large one, satisfies the desired properties and what is the proportion of matrices involved in the current identification problem that would satisfy these properties. In other words, the benefits of exploring that option are far from being obvious.

The special cases include, but may not be limited to:

1. When bipartite graph G, having A as its adjacency matrix, is planar (see section 2.2.3 and 4 for more details on that interpretation). In that case, it is possible to compute exactly the number of perfect matchings with $O(n^3)$ [15].
2. In mod 2, the $per(A) = det(A)$ is the same as the determinant. It is possible to compute the permanent mod 2^k in $O(n^{4k-3})$ for $k \geq 2$ [25].
3. $per(A) = 0$ that can be verified with a combination of permutations.

2.2 Approximation approaches

The development of exact polynomial-time algorithms for the permanent are for the moment nonexistent and remain very unlikely. For that reason, most of the effort in calculating the permanent are spent into finding a good approximation in polynomial-time. As mentioned above, from the precision perspective, there are two main classes of algorithms: the ones that produce fixed error and the ones that can't guaranty control on the error or at best can only bound it. Without surprise, the latter class, being less restrictive, includes more propositions than the first one.

2.2.1 Statistical methods

Statistical methods to approximate the permanent include defining an estimator, usually a Monte Carlo one, and run simulations with samples large enough to obtain a good estimate.

[2]The multiplicativity of the determinant allows to use the LU decomposition to reduce computation complexity. The permanent being a non-multiplicative function, this reduction is not possible.

More formally (from [14]), let Z be the approximation of $per(A)$. A randomized approximation is a Monte Carlo algorithm which takes A, ϵ and δ as input and output Z such that:

$$P((1-\epsilon)per(A) \leq Z \leq (1+\epsilon)per(A)) \geq 1-\delta \tag{2.3}$$

with $E(Z) = per(A)$ and $E(Z^2)$ finite, i.e. a bounded variance. The ultimate goal being to produce Z in polynomial-time. Such algorithm that runs in polynomial-time is referred as a Fully Polynomial-Time Randomized Approximation (FPRAS). The time required to reach ϵ, which is strongly related to the samples size, depends on the estimated variance. Larger the variance is, larger the samples need to be.

Typically, Monte Carlo estimators are known to produce high variance. And to reduce the variance, it requires to introduce elements in the samples that have an important impact in the permanent calculation.

The most interesting strategies found in the literature on permanent calculation are:

1. Markov Chain Monte Carlo (MCMC);
2. Importance sampling.

The first strategy consists in producing samples using a Markov chain. This Markov chain has a uniform stationary distribution over states that makes the permanent estimation more likely to converge quickly to a good estimate (within the ϵ bounds). The approach, which is described in 2.2.3 and in depth in 4, allows to control the error for a given confidence δ in polynomial-time, making it a FPRAS algorithm.

The second option consists in sampling from a distribution that over-weights the important elements in the permanent calculation and then adjust the estimate to account for having sampled from this other distribution. More details on that can be found in section 2.2.2.1.

2.2.2 Varying error

Algorithms belonging to this class provides an approximation factor of c^n, for different values of c, which sometimes also vary accordingly to matrix characteristics. In other words, for some types of matrices, as the size increases, the precision decreases. That being said, some methods described in this section, such as the importance sampling, can produce fixed error but for specific types of matrices only.

2.2.2.1 Importance sampling

Importance sampling is a form of Monte Carlo method designed to reduce the variance by sampling the important elements in the permanent computation with a higher biased probability and weigh the resulting element to remove the bias. Defining this bias is a very delicate operation that depends on the type of matrix involved in the permanent calculation. That is why there exist many importance sampling methods to estimate the permanent. These approaches relax the

condition on the accuracy, allowing it to vary as a function of the matrix characteristics (e.g. its size) for some types of matrices.

Because there is an important body of literature on that topic (relatively to the entire permanent approximation one), we will go quickly through it to briefly describe them and mention which are of interest for further investigation.

- Kuznetsov [18] presents an Importance Sampling (IS) algorithm for non negative matrices that produce estimation of the permanent that vary with n.
- Beichl and Sullivan [3] present an IS algorithm for 0-1 matrices in the context of the dimer covering problem.
- Smith and Dawkins [22] present multiple IS estimators and also review and compare with the KKLLL [14] and Barvinok [2] ones. This paper offers an intuitive insight on how to build a IS algorithm for permanent approximation.
- Chen et al. [7] present a sequential importance sampling for 0-1 matrices. Although the algorithm is restricted to 0-1 matrices, it offers a good review of the existing statistical approaches.
- Morelande [19] uses the approach proposed by Kuznetsov [18] in the context of joint data association.
- Kou and McCullagh [17] present also a sequential importance sampling but for symmetric positive definite matrices.

Because IS methods produce estimates with varying errors depending on the type of matrices explored, they are not the best fit for the problem at hand. But matrices involved in our context do have a common characteristic: duplicate rows and/or columns. It may be then possible to develop an IS algorithm that uses that characteristic to produce good estimates. However, that option would require significant additional research effort, both on the theory and the experimentation sides, that would go beyond the scope of this work.

2.2.2.2 Belief propagation

Perhaps the approach that attracted the most interest lately comes from the graphical model. It is currently the most active field of research for the permanent approximation.

The main idea is to re-write the permanent as a partition function over the space of all permutations of A. The partition function is then approximated by minimizing the Bethe free energy. The common approach used to minimize locally the Bethe free energy is the belief propagation algorithm. Consult [10] and [8] for the details. The latter provides the latest and tightest theoretical bounds while the first one provides an easier to read overview of the approach.

Results shown in [10] and [8] are somehow disappointing from an accuracy point of view. Although execution time is realistic for real applications and theoretical aspects of the approach are very promising, its lack of accuracy makes it not usable in a MSA identification context.

2.2.2.3 Theoretical bounds

Another strategy to estimate the permanent is to use one of the several theoretical bounds on the permanent. See [24] for a list of such bounds with experimental results.

These bounds were found to be too loose to be used in our context.

2.2.3 Fixed error

The main proposition for an approximation with control over the relative error is the MCMC approach.

It is worth mentioning at this point that computing the permanent of a 0-1 matrix A (i.e. $a_{ij} = \{0, 1\}$ for all $i, j = 1, ..., n$) is equivalent to counting the number of perfect matchings in the bipartite graph G having A as its adjacency matrix. This point of view is further developed in section 4.

The MCMC approach is based on the finding [13] that the existence of a FPRAS for the 0-1 matrix permanent is computationally equivalent to the existence of a polynomial time algorithm for sampling perfect matching from a bipartite graph almost uniformly at random. Broder [6] then designed a Markov chain whose stationary distribution is uniform over all perfect and near-perfect matchings of the graph, allowing to sample perfect matchings. Jerrum et al. [11] showed that the method was also working under relaxed conditions, leading to an $O(n^{26})$ algorithm to approximate the permanent of a 0-1 matrix with a fixed guarantied relative error.

The approach was then extended by Jerrum et al. [12] to arbitrary non-negative matrices, without any restriction on density. The resulting algorithm is $O(n^{11})$ which was later improved to $O(n^7)$ [4].

More details on these approaches can be found in section 4.

There was also a proposition made by Huber and Law [23] converging with $O(n^4)$ for very dense matrices. They propose an algorithm to generate weighted perfect matching exactly from their correct distributions. For non-dense matrices however, the performance of the algorithm is less interesting than the latest MCMC one.

Note that since Huber's work, nothing significant has been published in this area.

2.3 Algorithms selection

The main observation from the above review is that the time required to compute the permanent is roughly proportionally inverse to the obtained precision. This makes the first 3 requirements listed in section 1 incompatible.

In this work, we tried to push this limit by taking advantage of modern computer architecture allowing easy multi-threading. We decided to focus on methods allowing to calculate or approx-

imate the permanent with a fixed error (blue boxes in Figure 2.1) while trying to exploit the availability of multiple cores. This approach is not explicitly explored in the literature. The usual strategy aims to a reduction of the number of operations and/or tightening the approximation errors bounds.

We thus selected the algorithms providing the best guarantee on the error while having potential for parallelization, namely the Ryser algorithm and the MCMC. Section 3 shows the selected strategy to parallelize the Ryser algorithm and section 4 presents the MCMC approach.

This page is intentionally left blank.

Part 3

Ryser Algorithm

For the implementation of the Ryser algorithm, the notation of the CodeProject[1] was used:

$$per(A) = \sum_{t=0}^{n-1}(-1)^t \sum_{X \in \Gamma_{m-t}} \prod_{j=1}^{n} r_j(X), \tag{3.1}$$

where Γ_{m-t} is the set of all submatrices of A created by removing t columns, or otherwise said, all submatrices of A having $n-t$ columns. The element $r_j(X)$ is the sum of all elements on row j of the submatrix X.

3.1 Sequential version

The sequential implementation of the Ryser algorithm flow diagram is detailed in Figure 3.1. Input data and variables description are detailed below:

Input

- *Matrix[n][n]* (n = matrix size)

Variables

- *sumOfRows[n]*: One dimension table with the size n. Used to store the current iteration sum of each row.
- *currentRow*: The current row under summation.
- *productOfSums*: Used to store the product of all row sums.
- *currentIteration*: Current iteration to compute.
- *tempPerm*: Used to store the temporary value of the permanent computation.

[1] `http://www.codeproject.com/Articles/21282/Compute-Permanent-of-a-Matrix-with-Ryser-s-Algorit`

- *Permanent*: Final result of the permanent computation.

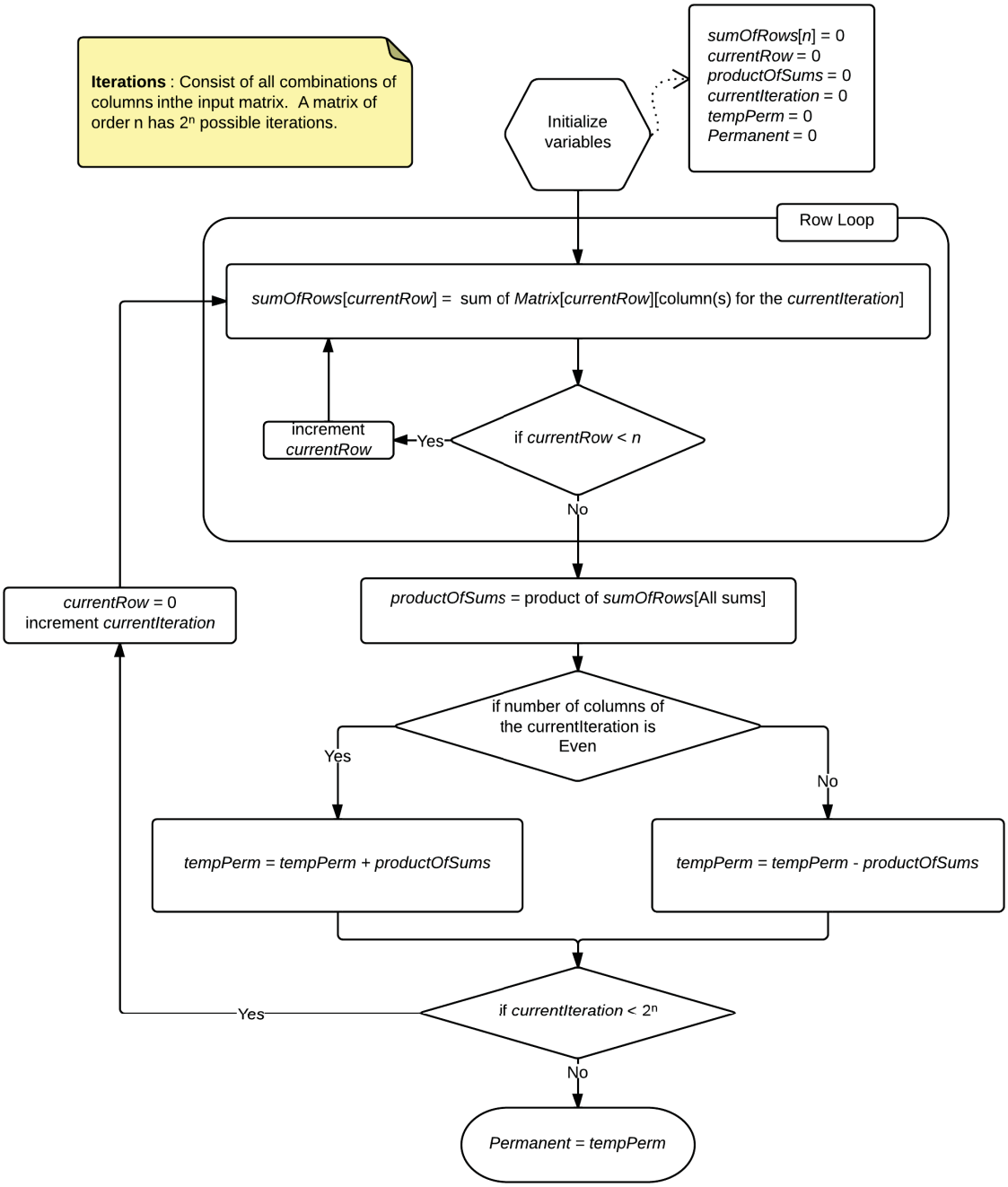

Figure 3.1: Ryser sequential algorithm.

To optimize each iteration, computing each time a different combination of columns, the Gray code [1] binary numerical system have been used. This binary representation has been used because only one bit changes at each increment of the Gray code. This helps minimizing the access to the matrix data in Random-Access Memory (RAM) memory, resulting to a gain in computing

Table 3.1: Matrix three by three with index.

0	1	2
3	4	5
6	7	8

speed. It is important to know that RAM memory access is costly in time compared to cache memory access. Matrix data access is reduced because when reading a value the following data is also loaded in cache memory, this data then is available for the next operation because it is still in cache memory. A special care has also been taken to loop over matrix index in the order the matrix is stored in memory (memory layout). By doing so we also improved access to data and then reduced the computing time. For instance, reading from left to right (index 0 to 2 for row one) in the matrix represented in Table 3.1 is quicker than reading from top to bottom.

3.2 Multi-Threaded version

The efficiency gained by performing a parallelized algorithm greatly depends on the design of the algorithm itself. Some are less prone to parallelism because it would add too much workload to the processor due to inter-process communication.

Several iterations were performed to test and choose the best parallel implementation for the Ryser algorithm before getting a parallel optimized solution. The parallel solution has been designed to reduce as much as possible the inter-process communication. Although the algorithm runs on a single computer, it is easily extensible to multiple computers.

Parallelization implementation is detailed in Figure 3.2 and input data and variables description are detailed below:

Input

- *Matrix[n][n]*: Shared with all process (n = matrix size).

Variables

- *sumOfRows[n]*: One dimension table with the size of n. Used to store the current iteration sum of each row. Each process has one copy.
- *currentRow*: The current row under summation. Each process has one copy.
- *productOfSums*: Used to store the product of all rows sum. Each process has one copy.
- *currentIteration*: Current iteration to compute. Each process has one copy.

- *nbProcess*: Number of processes available on the current computer. Global to all processes.
- *processID*: Identification of a started process. The maximum *processID* value is *nbProcess*-1. Each process has one copy.
- *nbIteration*: Number of iteration to do in regard of the number of process. One copy for each process.
- *tempPerm[nbProcess]*: Used to store the temporary value of the permanent computation of every processes. Accessible, read only, by all processes.
- *Permanent*: Final result of the permanent computation.

The matrix columns are added and subtracted constantly throughout the iterations. If we count the number of times each column is visited, we note that the last column is less visited. This offers the possibility for parallelization. For a matrix of size 20 ($n = 20$), the first column is visited 524,288 times while the last column is visited only one time. Each of the columns has been visited $2^{((n-1)-j)}$ times (where j is the index of the column). Computation can be easily distributed over multiple processes using the latest columns which form the latest combinations possible. To do so, the algorithm pre-calculates these combinations and then results are distributed to previously initialized processes. The Ryser algorithm is thus executed by each process without calculating the latest column that has already been pre-calculated. In doing so, we avoid repeating calculation, and therefore reduce the execution time.

To avoid too many context-switches by the operating system, which slow down the execution, it is important to limit the inter-process communications. In the proposed approach, processes have very little communication with each other. In fact, each process communicates only with the main process. Initially each process is given a copy of the input matrix and when all processes finish their computation they save their final result in a shared variable (an array). When all processes had been completed, all results are merged to obtain the permanent.

The implemented parallelization has however a limitation: it depends on the number of processors available. Indeed, the number of processes available must be a power of 2 (2,4,8,16,32,64 ...).

3.3 Reduced Version

Matrices involved in the identification of tracks/targets for MSA usually have duplicated rows and columns. In [21], the author identifies a reduction opportunity with the Ryser algorithm using the Gray-code order.

We adapted this algorithm to implement it within the multi-threaded framework presented in section 3.2. This section focuses solely on the implementation. For details about the algorithm itself, refer to [21].

Figure 3.3 presents this reduced version of the multi-threaded Ryser algorithm for matrices with duplicated columns. The resulting algorithm is similar to multi-threaded one (see Figure 3.2), the additional elements are indicated in red in Figure 3.3.

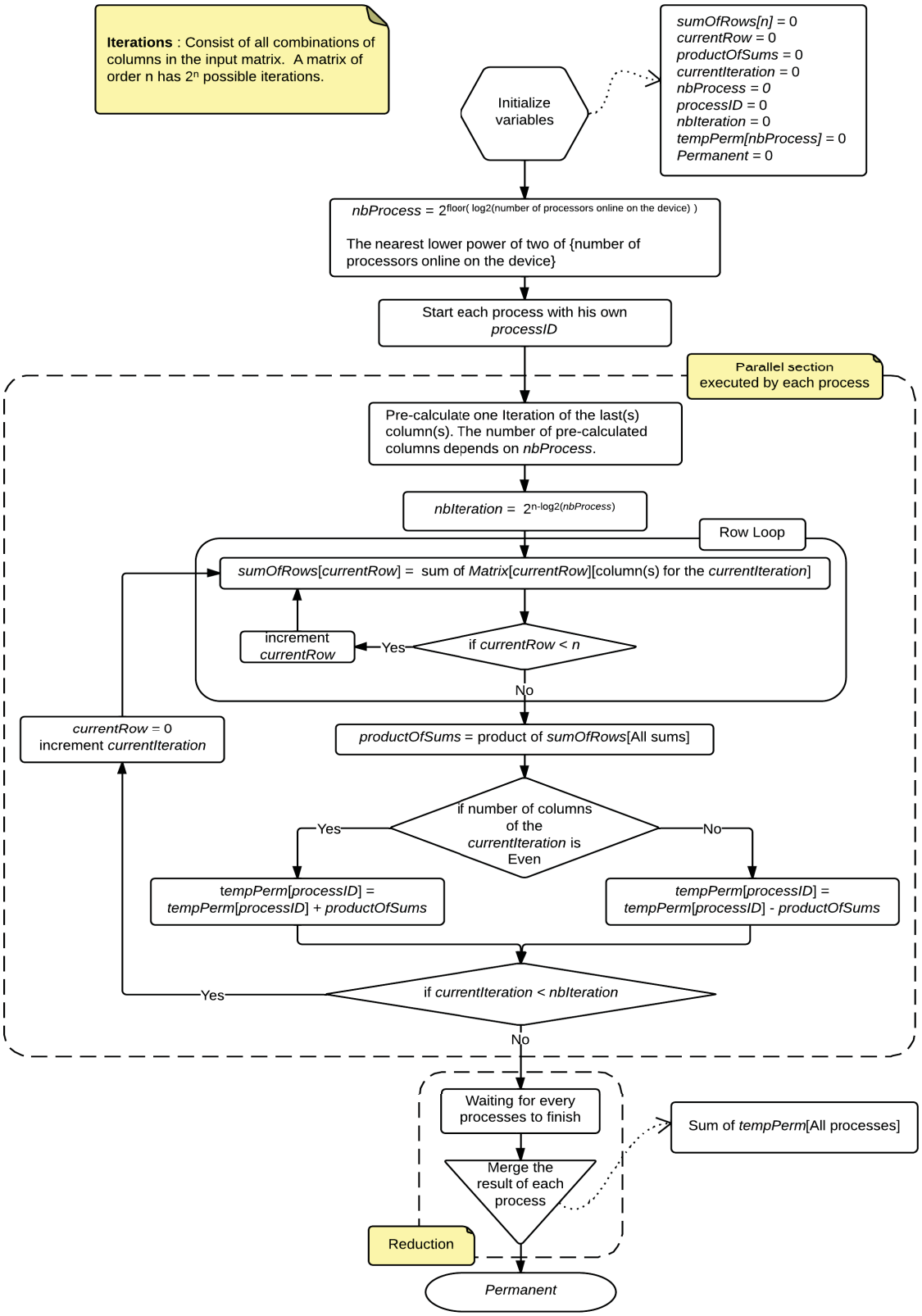

Figure 3.2: Ryser parallellized algorithm.

The required input and algorithm variables are the same than for the multi-threaded Ryser algorithm (see section above), with an additional input:

- *duplicatedColumns[n]*: a vector identifying the duplicated columns. For instance, if we have a 3 by 3 matrix with the last 2 columns identical, then $duplicatedColumns[n] = [1, 2, 0]$. If we have a 5 by 5 matrix with the 3 first columns identical and the 2 last identical, then $duplicatedColumns[n] = [3, 0, 0, 2, 0]$.

The multi-threaded Ryser algorithm can be divided in 4 main blocs:

1. Initialize the variables;
2. Setup the parallel processing by preparing the tasks;
3. Launch the parallel processes (algorithm execution in parallel);
4. Merge the results of each process.

To simply wrap the initial implementation in the parallel framework would have require to distribute the execution over the independent processes (bloc 3). However, it was not possible because there is information passed between iterations. So we had to break communication dependencies part of the initial implementation and push it in bloc 2.

Two processes have been removed from the algorithm execution and moved to the parallel processing setup (bloc 2):

1. identification of the unique combinations of duplicated columns and
2. computation of the weights.

The unique combination of duplicated columns is split and the associated weights calculated prior to launch the parallel processes. The number of parallel processes launched is the number of unique combinations of repeated columns. Each process calculates the result of the combination with its associated weight. Then each process executes the Ryser's algorithm avoiding duplicated columns. Once parallel processing is completed, all results are combined to produce the permanent.

The implementation of the algorithm pre-calculates the maximum number of unique combinations even if the number of processes available on the running computer is limited. It means that if there is too much combinations, the algorithm will spawn too many processes for the available units, resulting in a slowdown due to processes concurrency. This aspect should be addressed to improve robustness.

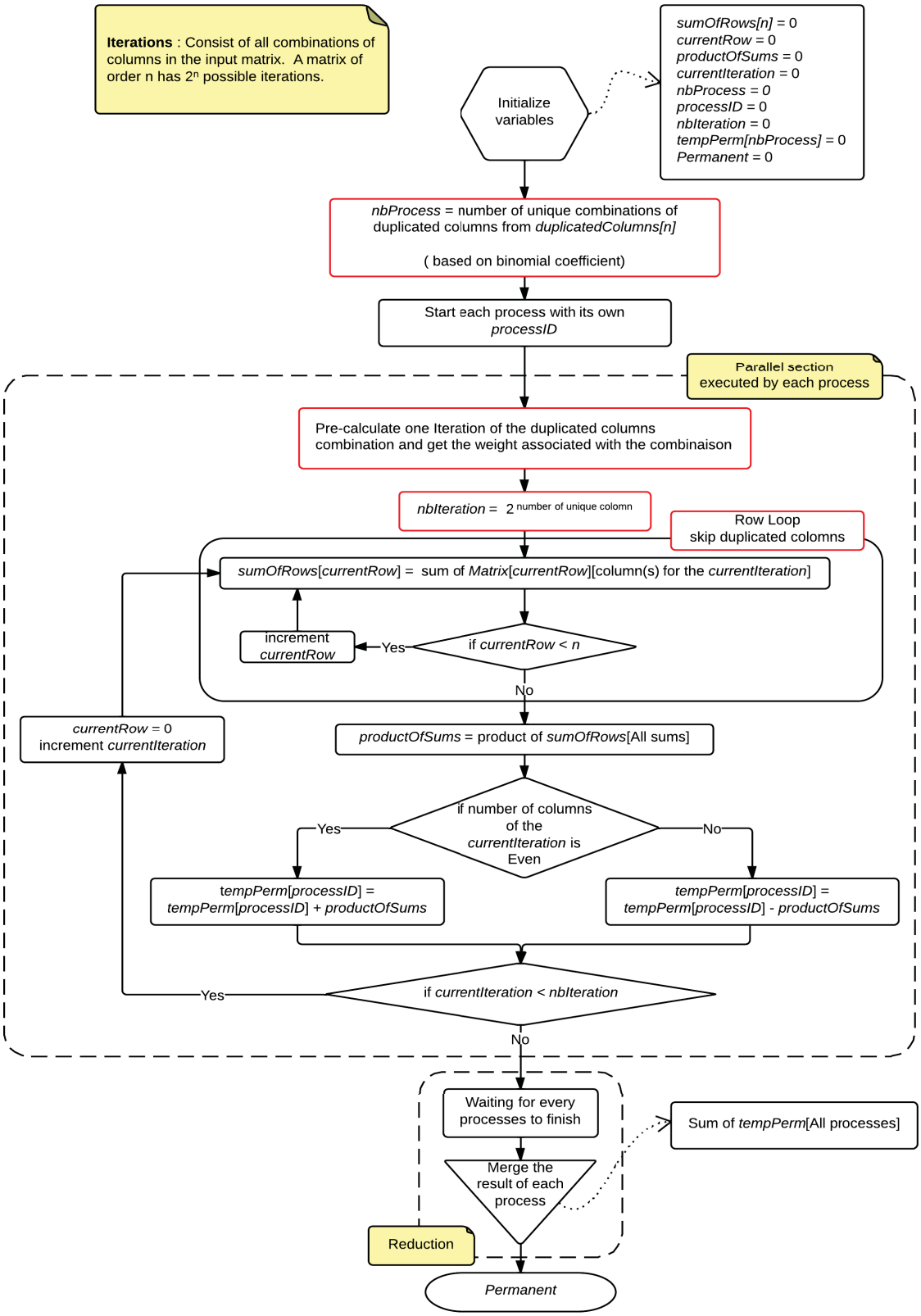

Figure 3.3: Reduced Ryser parallellized algorithm.

This page is intentionally left blank.

Part 4

Markov-Chain Monte-Carlo Methods

If A is 0-1 square matrix, then $per(A)$ is the number of perfect matchings in G, where A is the adjacency matrix of the graph G. A perfect matching of G is a graph where all n nodes are matched to a single node with an edge belonging to G.

For instance, consider

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{4.1}$$

for which $per(A) = 2$. The graph G and its 2 perfect matchings are illustrated in Figure 4.1

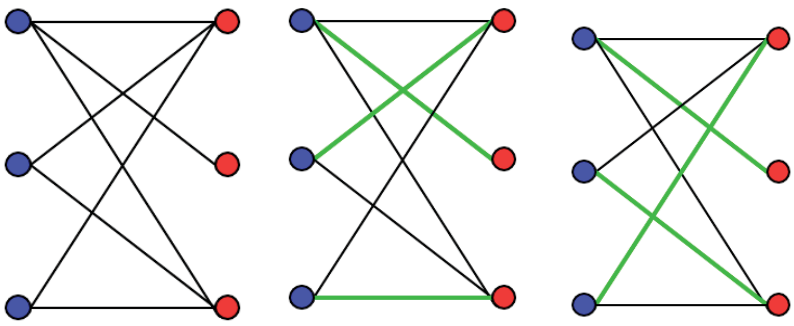

Figure 4.1: The bipartite graph with adjacency matrix A defined in 4.1 and its 2 perfect matchings (in green).

Let us call $M_k(G) \equiv M_k$ the set of all matchings of size k of G. Using that notation, M_n is the set of all perfect matchings and M_{n-1} the set of near perfect matchings. A matching of size k is a sub-graph of G where exactly k nodes are matched to a single node.

Computing the permanent of A is thus equivalent to computing $|M_n(G)|$.

4.1 Basic approach

In the basic approach, to approximate $|M_n(G)|$, the following relation is used:

$$|M_n| = \frac{|M_n|}{|M_{n-1}|}\frac{|M_{n-1}|}{|M_{n-2}|}...\frac{|M_2|}{|M_1|}\frac{|M_1|}{|M_0|} = \prod_{i=1}^{n} R_i, \tag{4.2}$$

where $R_i = \frac{|M_i|}{|M_{i-1}|}$ and $|M_0| = 1$.

The basic approach estimates each R_i for $i \in [1, n]$ with a MCMC simulation.

Suppose we had an estimator S_i that takes a match $M \in \Omega = M_i \cup M_{i-1}$ as input and output $M \in M_i \cup M_{i-1}$ with *uniform probability*. With a large enough sample of size S, we could count the number of matchings belonging to M_i and M_{i-1} to get an estimation of R_i:

$$R_i \sim \frac{\#M \in M_i}{\#M \in M_{i-1}} = \frac{\#M \in M_i}{S - \#M \in M_i}. \tag{4.3}$$

To explain this intuitively, let us consider a bag containing 8 red and 2 blue marbles, each one having the same size, shape and weight. Suppose one had to estimate the ratio of red over blue marbles (which is 4) without being able to look inside the bag. An option would be to repeat several times the following game: pick one marble, note its color and put it back in the bag. If this game is played long enough, the number of red observations over the blue ones would provide a good estimate of the ratio.

Broder [6] proposed a Markov chain for the estimator S_i. This chain, illustrated in Figure 4.2, is irreducible and aperiodic. It thus converges to its stationary distribution π over Ω, regardless of the initial state and the stationary distribution is uniform over Ω. Moreover, there exists a minimum number of iterations T such that the chain converges to an almost uniform state.

This algorithm is a FPRAS if each ratio R_i is polynomially bounded, which is the case for dense matrices but not necessary for non-dense matrices.

4.1.1 Implementation

To implement this method, it is easier to work with matching expressed as a matrix instead of a graph. But for clarity, the algorithm is described in this report using the graph notation.

The final output is the product 4.2, which can be re-written as:

$$|M_n| = |M_1| \prod_{i=2}^{n} R_i = (\#\text{edges in G}) \prod_{i=2}^{n} R_i \tag{4.4}$$

$|M_1|$ is the number of edges in G since a M_1 matching is a sub-graph of G having only one pair of nodes matched.

We start by estimating R_2, which requires initiating S Markov chains over $\Omega_2 = M_1 \cup M_2$. The chain takes as input a matching M from either M_1 or M_2 and output a matching from Ω_2. From

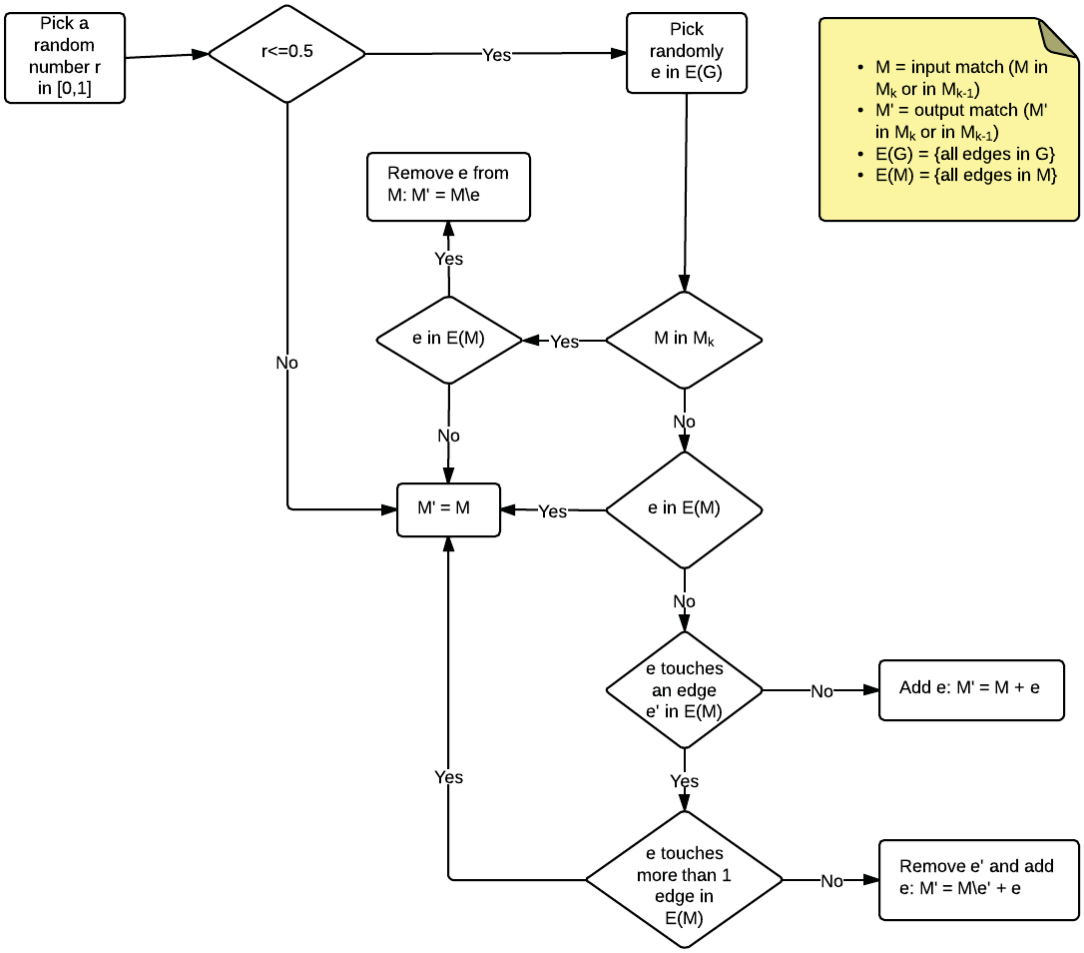

Figure 4.2: Flow diagram for the Markov chain in the basic approach.

an implementation point of view, it is easier to build a M_1 matching. Indeed, you remove all edges from G except one. Let us call 'M_1' this input matching. The MCMC simulation to estimate R_2 can thus be summarized as:

$$\begin{aligned} M^{(1,1)} = M^1 \curvearrowright M^{(2,1)} \curvearrowright ...M^{(T,1)} \in \Omega_2 \\ M^{(1,2)} = M^1 \curvearrowright M^{(2,2)} \curvearrowright ...M^{(T,2)} \in \Omega_2 \\ ... \\ M^{(1,S)} = M^1 \curvearrowright M^{(2,S)} \curvearrowright ...M^{(T,S)} \in \Omega_2 \end{aligned}$$

Each change of state in the Markov chain ($\curvearrowright$) is described by the flow diagram in Figure 4.2.

Once the simulation is finished, we estimate the ratio:

$$R_2 \sim \frac{\sum_{i=1}^{S} I_{M_2}(M^{(T,i)})}{S - \sum_{i=1}^{S} I_{M_2}(M^{(T,i)})},$$

where $I_{M_2}(M)$ is 1 if $M \in M_2$ and 0 otherwise. In other words, R_2 is estimated by the number of M_2 matchings over the M_1 matchings. Note that a matching is in M_i if it has i edges.

Once R_2 is computed, the next step is to estimate R_3 with a similar mechanism. The input matching M^2 is one of the $M^{(T,i)} \in M_2$ matchings produced in the previous MCMC run:

$$\begin{aligned} M^{(1,1)} = M^2 \curvearrowright M^{(2,1)} \curvearrowright ...M^{(T,1)} \in \Omega_3 \\ M^{(1,2)} = M^2 \curvearrowright M^{(2,2)} \curvearrowright ...M^{(T,2)} \in \Omega_3 \\ ... \\ M^{(1,S)} = M^2 \curvearrowright M^{(2,S)} \curvearrowright ...M^{(T,S)} \in \Omega_3 \end{aligned}$$

and $R_3 \sim \sum_{i=1}^{S} I_{M_3}(M^{(T,i)})/(S - \sum_{i=1}^{S} I_{M_3}(M^{(T,i)}))$.

The rest of the ratios R_i ($i \in [4, n]$) are estimated similarly and the permanent is obtained with the product (4.4).

For this particular Markov chain, we could not find the estimated values for T nor S. However, S is at least of $O(n^2\epsilon^{-2})$ to make sure that the algorithm approximates, within a factor of $1 \pm \epsilon$, the count of perfect matchings, i.e. the permanent of the 0-1 matrix A.

4.2 Arbitrary non-negative matrices

The first FPRAS for arbitrary (meaning no restriction on the ratios R_i) non-negative matrices was proposed by Jerrum et al. [11] with a running time of $O^*(n^{26})$[1]. It was improved to $O^*(n^{11})$ [12] and then to $O^*(n^7)$ [4]. We present here the latter version.

Suppose a graph (thus a 0-1 matrix) having exponentially more near perfect matchings than perfect matchings. The resulting ratio R_n would be exponentially small, i.e. perfect matchings would have

[1]The notation O^* hides the log factor and the dependence on the error parameter.

insufficient weight in the stationary distribution, and the Markov chain presented in the previous section could not be used to sample perfect matchings in polynomial time.

The proposed solution is to weigh the states of the Markov chain so that the weighted ratio R_n is always polynomially bounded.

Let $M_{n-1}(u, v)$ be the set of all near perfect matchings having unmatched holes u and v. For instance, Figure 4.3 illustrates the 2 near perfect matchings with holes u and v for the bipartite graph with adjacency matrix A defined in 4.1.

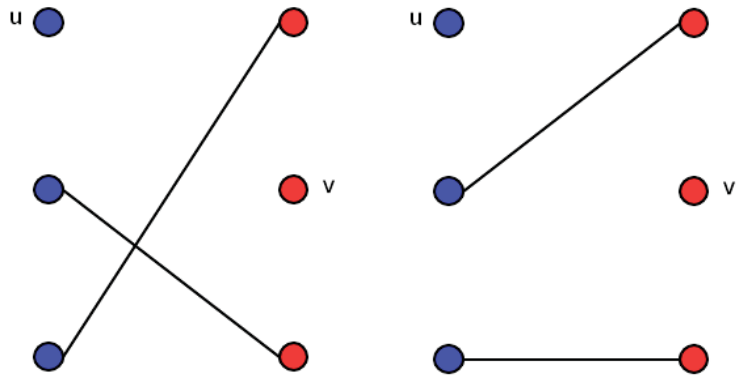

Figure 4.3: Near perfect matchings with holes u and v for the bipartite graph with adjacency matrix A defined in 4.1.

Since there are n^2 possible pairs of holes u, v, $\Omega_n = M_n \cup M_{n-1}$ contains $n^2 + 1$ sets each being possibly of different size: n^2 $M_{n-1}(u, v)$ sets plus the M_n set.

It was proposed in [11] a modified Markov chain with a stationary probability such that the total probability of all near perfect matchings having holes (u, v) is equal to the total probability of all perfect matchings. Otherwise said, any hole pattern, including without hole (perfect matchings), is equally likely under the stationary distribution.

To design such a Markov chain, weights need to be introduced so that the perfect matchings have equal weight in the stationary distribution than the near perfect ones. This is illustrated by Figure 4.4 for a matrix of size 3. These weights, that we will refer to be the *ideal* weights w^*, are defined by:

$$w^*(u, v) = |M_n| \,/\, |M_{n-1}(u, v)| \tag{4.5}$$

Since these weights are a function of the total number of perfect matchings, computing them is as difficult as counting the perfect matchings. This is why the ideal weights need to be estimated.

The algorithm contains two main blocs:

1. Estimate the weights via a MCMC strategy.
2. Estimate $|M_n|$ with an additional MCMC strategy using the weights.

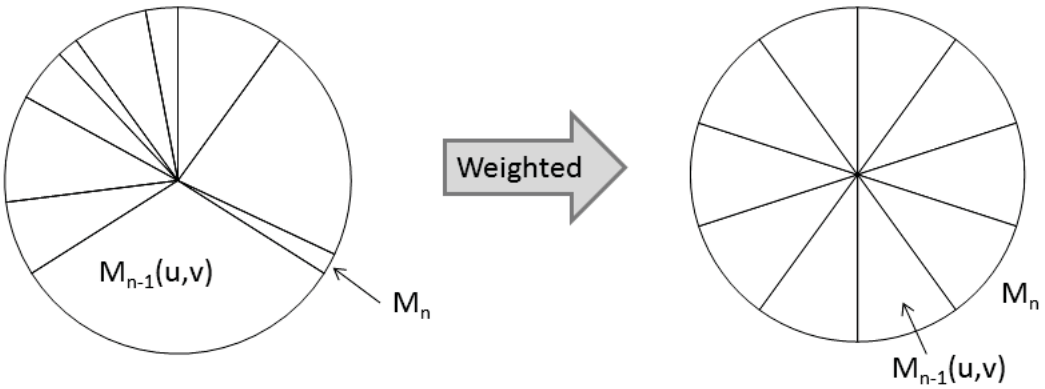

Figure 4.4: Weighted states for the modified Markov chain.

The first bloc is used to produce the weights to sample the perfect and near perfect matchings while the second one to sample them to estimate $|M_n|$.

4.2.1 Weights estimation

Unlike the basic approach, to estimate the weights for each pair of holes, we only need to work with $\Omega_n = M_n \cup M_{n-1}$. The idea is to iterate, via multiple MCMC simulations, until convergence of the weights to their *ideal* value. Iterations or phases are defined by a parameter called *activity*, which is a weight, varying from 1 to $1/n!$, on each on edge. The main advantage to introduce such parameter is that it allows to work on the complete bipartite graph G_C (the graph where all nodes are connected or equivalently, a matrix of size n filled with ones).

The activity of an edge e is

$$\lambda(e) = \begin{cases} 1, & e \in G \\ \lambda_k, & \text{otherwise.} \end{cases} \tag{4.6}$$

where λ_k decreases from 1 to $1/n!$ over the K successive phases. Note that an edge e in a bipartie graph can be defined by the two nodes (u, v) it connects:

$$\lambda(u, v) = \begin{cases} 1, & (u, v) \text{ are connected in G} \\ \lambda_k, & \text{otherwise.} \end{cases} \tag{4.7}$$

This definition allows to define the activity of a matching M and of a set S of matchings:

$$\lambda(M) = \prod_{(u,v)\in M} \lambda(u, v), \lambda(S) = \sum_{M\in S} \lambda(M).$$

For each pair of nodes (u, v), or equivalently for each edge e they are connected with, there is a positive real number $w(u, v)$ called the weight, for a total of n^2 such weights for the graph. Given

weights w, the weight of a matching $M \in \Omega$ is:

$$w(M) = \begin{cases} \lambda(M)w(u,v), & M \in M_{n-1}(u,v) \\ \lambda(M), & M \in M_n \end{cases} \tag{4.8}$$

and the weight of a set S of matchings is $\sum_{M \in S} w(M)$. We can link that definition with the ideal weights w^*, defined by (4.5), we are trying to compute by

$$w^*(u,v) = \frac{\lambda(M_n)}{\lambda(M_{n-1})} \tag{4.9}$$

because

$$\lambda(M_n) = \sum_{M \in M_n} \lambda(M) = \sum_{M \in M_n} \prod_{(u,v) \in M} \lambda(u,v) = \sum_{M \in M_n} 1 = |M_n| \tag{4.10}$$

and similarly

$$\lambda(M_{n-1}(u,v)) = |M_{n-1}(u,v)|\,.$$

With these definitions in hand, the algorithm to compute the ideal weights consists in updating the weights $w_{k+1}(w_k, \lambda_k)$ through K successive phases of MCMC runs until K is large enough to have $w_{k+1} \sim w^*$.

The Markov chain used for the convergence of the weights is the same as the one described in Figure 4.2, except that the state M' is accepted with probability

$$min(1, w(M')/w(M)) \tag{4.11}$$

and that the states are always $\Omega_n = M_n \cup M_{n-1}$.

The stationary distribution of this Markov chain is

$$\pi(M) = w(M) / \sum_{M \in \Omega} w(M). \tag{4.12}$$

From equations 4.12 and 4.9, we then have

$$w^*(u,v) = w(u,v) \frac{\pi(M_n)}{\pi(M_{n-1}(u,v))}. \tag{4.13}$$

To update the weights, S Markov chains are executed during T iterations. Since we are working only on $\Omega_n = M_n \cup M_{n-1}$, the output of each chain will be a matching $M \in \Omega_n$. For each chain, we count the number of matchings belonging to each $n^2 + 1$ class M_n and the n^2 $M_{n-1}(u,v)$ ones. These counts are used to update the n^2 weights $w(u,v)$:

$$w_{k+1}(u,v) = w_k(u,v) \frac{\sum_{i=1}^{S} I_{M_n}(M^{(T,i)})}{\sum_{i=1}^{S} I_{M_{n-1}(u,v)}(M^{(t,i)})}.$$

4.2.2 Permanent estimation

The permanent is also estimated via a telescopic product, but using the weights computed at each phase:

$$|M_n| = \frac{|M_n|}{w_K(\Omega)} \frac{w_K(\Omega)}{w_{K-1}(\Omega)} ... \frac{w_1(\Omega)}{w_0(\Omega)} w_0(\Omega)$$

Since there are $n!$ perfect matching in G_C, $w_0(\Omega) = n!(n^2+1)$ which leads to

$$|M_n| = n!(n^2+1) \frac{|M_n|}{w_K(\Omega)} \frac{w_K(\Omega)}{w_{K-1}(\Omega)} ... \frac{w_1(\Omega)}{w_0(\Omega)} = n!(n^2+1) \frac{|M_n|}{w_K(\Omega)} \prod_{j=1}^{K} R_j.$$

Each ratio R_j is estimated via the following experiment:
Run the Markov chain used for the weight estimation with $w = w_j$ and for T' times, output M and compute $w_{j+1}(M)/w_j(M)$. Repeat this experiment for S' times and take $R_j \sim \sum_{S'}(w_{j+1}(M)/w_j(M))/S'$.

The last ratio $|M_n|/w_K(\Omega)$ is estimated with the following experiment:
Run the Markov chain used for the weight estimation with $w = w_K$ and for T' times, output M and compute $I_{M_n(G)}(M)$. Repeat this experiment for S'' times and take $|M_n|/w_K(\Omega) \sim \sum_{S''}(I_{M_n(G)}(M))/S''$.

4.2.3 Implementation

Initial values for the weights estimation are $w(u,v) = n$ and $\lambda_0 = 1$. We found two very similar ways to update λ_k until it reaches $\lambda_K = 1/n!$, both requiring a total of $K = O(nlog^2 n)$ phases. See [4] and [5] for both versions.

Convergence of the weights and rapid mixing were demonstrated in [4] to be reached with $S = O(n^2 logn)$ and $T = O(n^5 logn)$ which can be improved to $O(n^4 logn)$ with warm starts. As for the permanent approximation, the required S', S'' and T' values are $O(K\epsilon^{-2})$,$O(n^2\epsilon^{-2})$ and $O(n^4 logn)$ respectively for a total of $O(KS' + S'')$. The total number of operations, using warm starts, is thus of $O(n^7 log^4 n + n^6 log^5 n\epsilon^{-2})$.

A *warm start* consists in using the output a Markov chain as input for the next. In this case, it means to use the set of matchings produced at the end of a phase as the initial matchings for the next phase. This mechanism allows the chains to mix faster.

The analysis and implementation efforts for this algorithm did not produce a successful demonstration of this approach within the allocated time and budget for this project. Some of the hurdles included:

- Many variables to initiate and one of them may have been initiated wrong.
- Many dependencies among the different processes (the output of a simulation being the input of another one), which increases complexity and usually greatly impact stability.

- There is no result in any of the 3 papers on the topics. It indicates that authors are more interested in the theoretical aspects of the algorithm than its performance. This orientation may lead to a lack of rigour in the definition of the algorithm parameters.

As we don't have an implementation of the algorithm for 0-1 matrices, we are not presenting here the extension to $\mathbb{R}^+$ matrices. See [4] for the details. The extension seems to only impact the weights estimation and impact is limited. However, it may hide additional complexity that can't be detected without implementing it.

This page is intentionally left blank.

Part 5

Performance Assessment and Recommendations

Performance have been measured using two strategies:

1. using different data types for the input matrix,
2. using different version of the Ryser algorithm: sequential and multi-threaded, described respectively in sections 3.1 and 3.2.

The focus of the performance measurement was the execution time and the relative error. Appendix A provides the details of the machine used for testing.

The sequential version of Ryser is referred as *Seq* while *Par* represents the parallel [1] one. The machine used for testing allowed the parallel version to have access to a maximum of 8 processes. Data types tested were:

- Double,
- Float80bits,
- GMP.

See Table 5.1 for data types details.

5.1 Execution Time

We used matrices of ones[2] for this series of tests. Execution time results presented in Figure 5.1 are for permanent calculation with the Ryser algorithm.

[1] The term *parallel* in this context is the same as multi-threaded because only one machine is involved.
[2] A matrix of ones is a matrix where every element is equal to one.

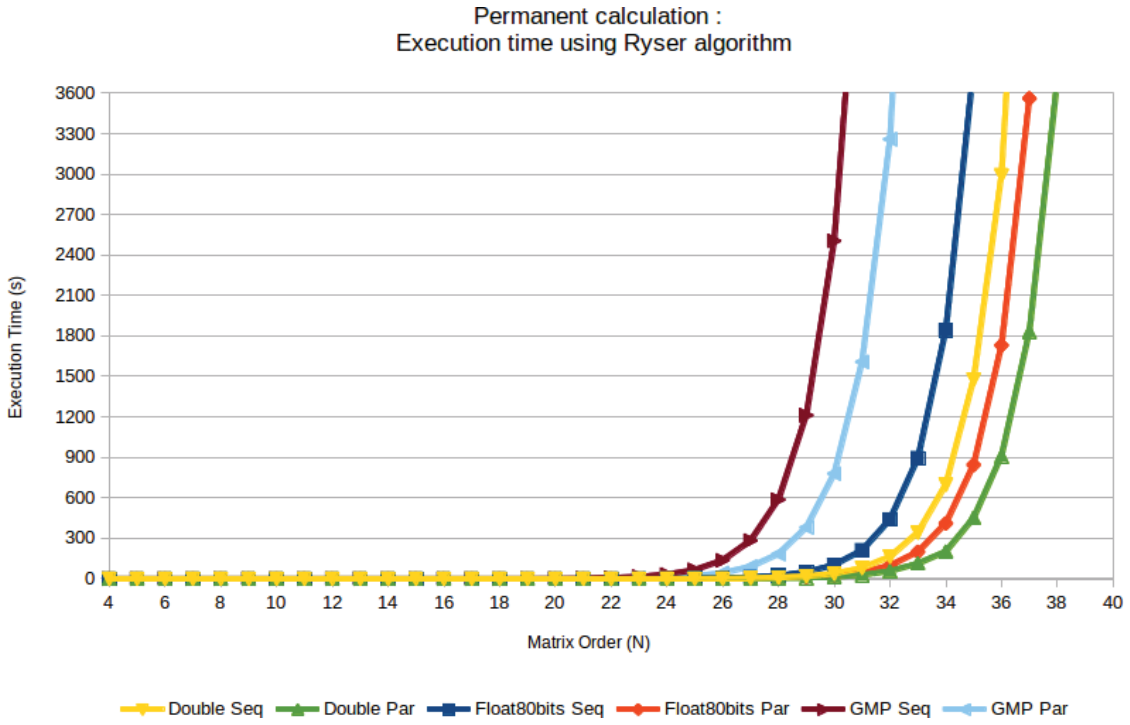

Figure 5.1: A comparison of the Ryser algorithm execution time with different data types and algorithm versions.

The first observation is that for each data type, the parallel implementation is faster. The best performance is reached when using the *Double* data type with the parallel implementation, but as shown in section 5.2, this type produces results with limited precision. The worst performance in terms of execution time is reaches when using the *GMP* data type with the sequential version. This is not a surprise, since using higher precision data type requires more memory and more operations by the Central Processing Unit (CPU). Globally we can see that all tests reach a critical point, in terms of matrix size, where the execution time rises exponentially. With the Ryser algorithm this critical point is already reached, with our hardware, when the matrix size is 40.

We could not run matrices of large size (e.g. $n = 1,000$), because of the hardware available was limited. But doing so we can expect an increase in RAM memory access due to the limited size of the cache memory compared to the size of the matrix. That problematic will result in an increase of the execution time.

The speedup is another performance metric that is interesting to look at in Figure 5.2. We measured the speedup between the sequential and the parallel versions of the Ryser algorithm for the three data types. We can see that the speedup is in function of the matrix size. The data type that better performed is the *Float80bits* with a stable speedup of about 4.5. GMP data type speedup stabilizes at about 3.2 and the *Double* data type seems to stabilizes at 3.5.

We also compared the influence of the input matrix data values on the execution time. Tests results can be seen in Figure 5.3. These tests have been performed with the *Double* data type and the parallel version of the Ryser algorithm. First series of tests was done with matrices of

Table 5.1: Different data type used to measure performance.

Data type	**Size in memory (bits)**	**Exponent size(bits)**	**Mantissa size(bits)**	**Significant decimals**	**Implementation**	**Description**
Double Precision (Double)	64	11	52	15.95	libstdc++ (IEEE754)	IEEE754 standard of double precision, binary64
Extended Precision (Float80bits)	128(80 used)	15	64	18	libstdc++(Gcc x86_64)	Specific implementation of gcc compiler. May be different on another system. long double (80bits).
GnuMultiPrecision (GMP)	192	Inf	50	15	GMP Library	Mpf_class using default precision (50bits). Uses infinite mantissa for arithmetic function.

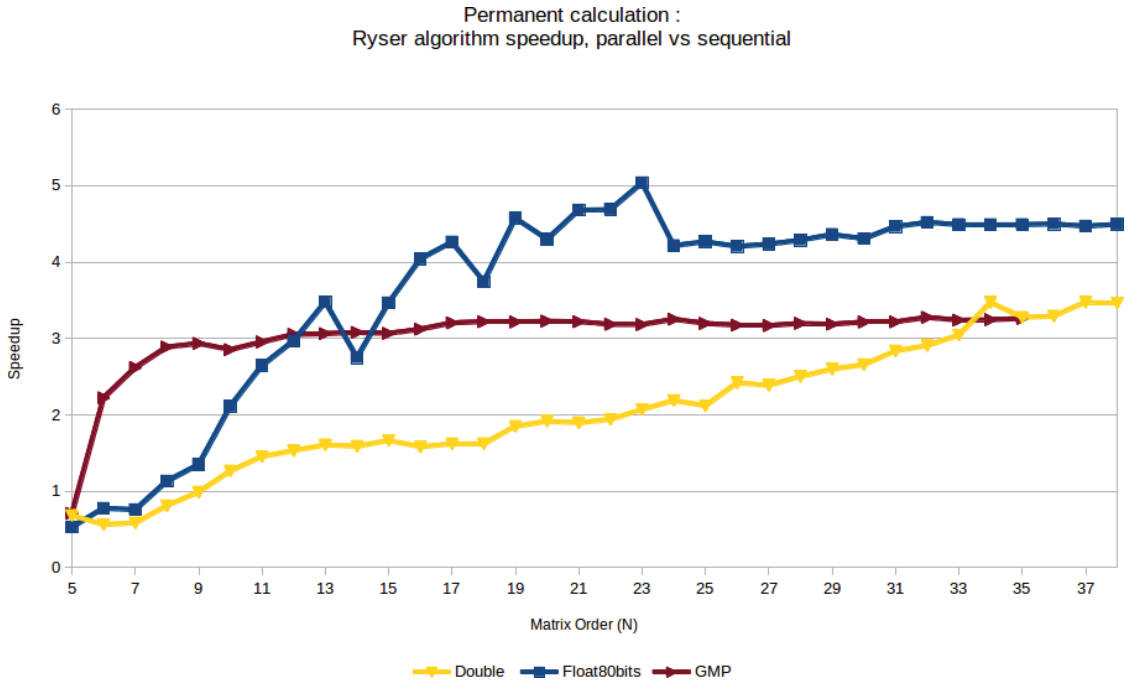

Figure 5.2: Ryser algorithm speedup gained using the parallel version over the sequential one.

ones (see *Double Par (all 1)*) and the second with matrices filled with random values from zero to one (*Double Par (0 to 1)*). No significant difference in execution time can be observed, **which is because there is no variation in the implementation of the algorithm related to data value.**

5.2 Precision

Precision results presented in Figure 5.4 show the relative error on permanent calculation with the Ryser algorithm. In order to compute the relative error, we used matrices of ones so the permanent value is $n!$.

Results show that the relative error rises quickly with an increase of the matrix size for data types *Double* and *Float80bits*, when using the sequential implementation of the Ryser algorithm. This can be explained by the fact that each process (one for the sequential implementation and many for the parallel implementation), needs to manage a larger number in a single variable resulting in a loss of precision. This larger number is obtained by the summations and multiplications required by the algorithm. In comparison, the *GMP* data type uses an infinite precision to do mathematical operation. The precision is only lost when the result of an operation is saved back in the GMP variable. This is why the *GMP* data type is better at handling error. *GMP* data type has also the particularity to behave the same with the sequential and the parallel algorithm. This can be explained by the fact that the *GMP* data type uses an infinite precision for arithmetic operations.

We see that all data types reach a point where the relative error rises exponentially. The exponent

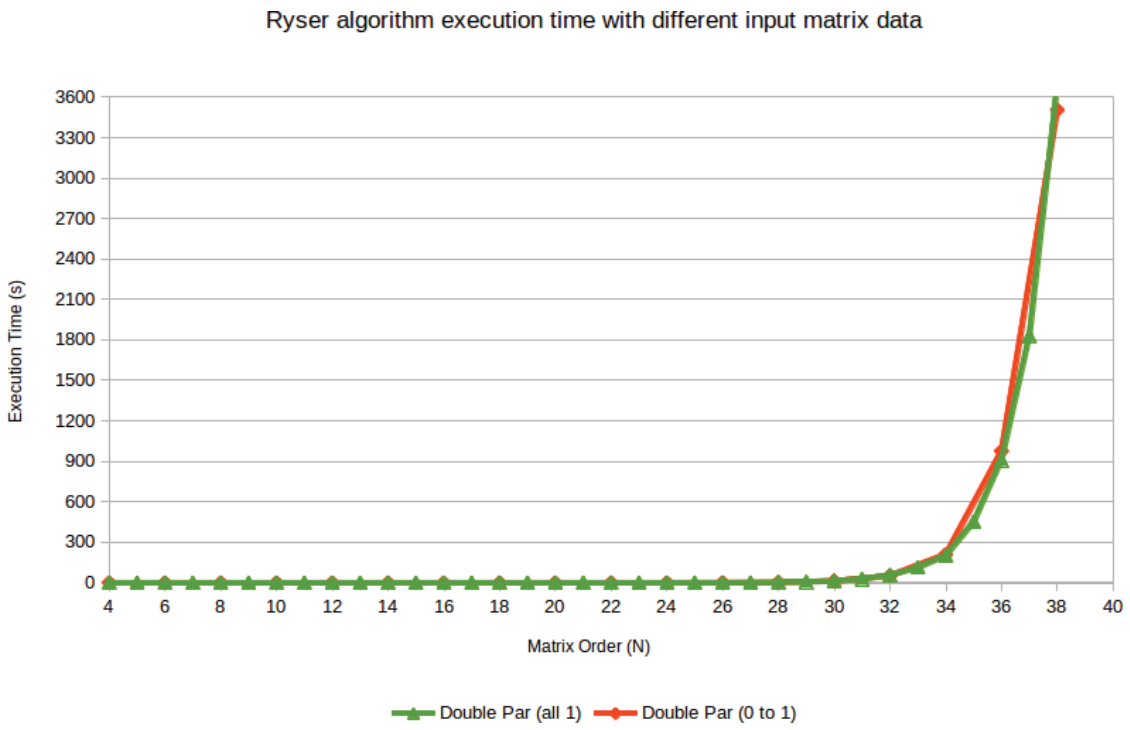

Figure 5.3: Influence of the matrix data values on the execution time for the parallel version of Ryser.

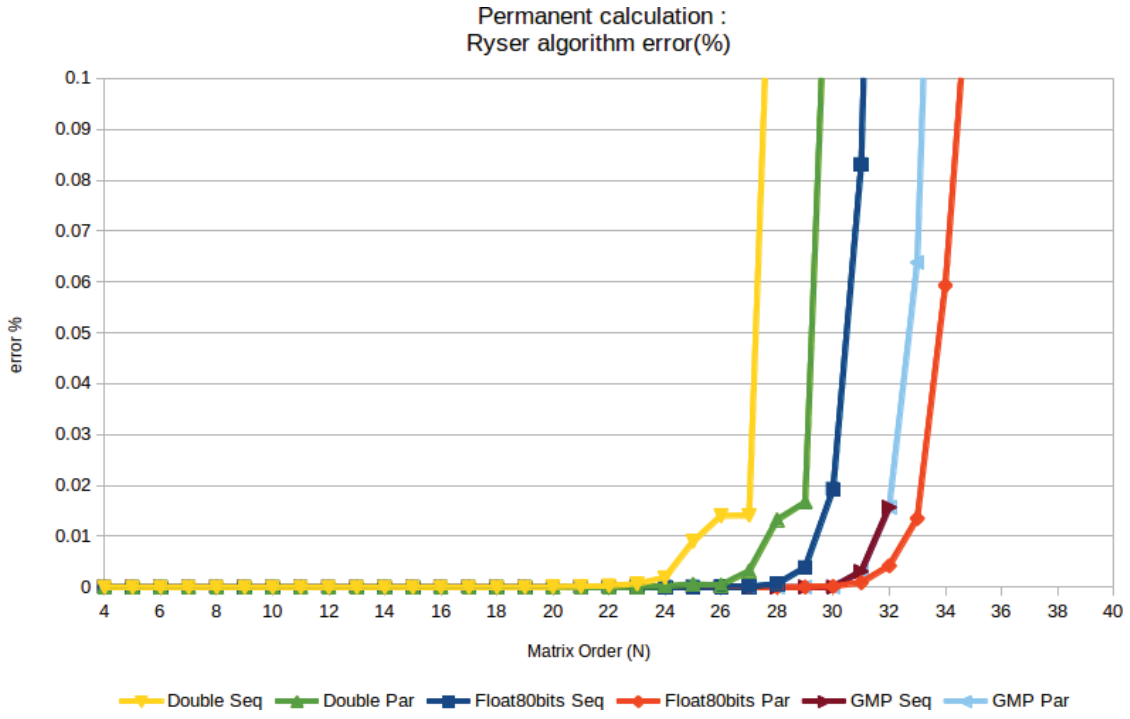

Figure 5.4: Relative error for different versions of the Ryser algorithm with different data types.

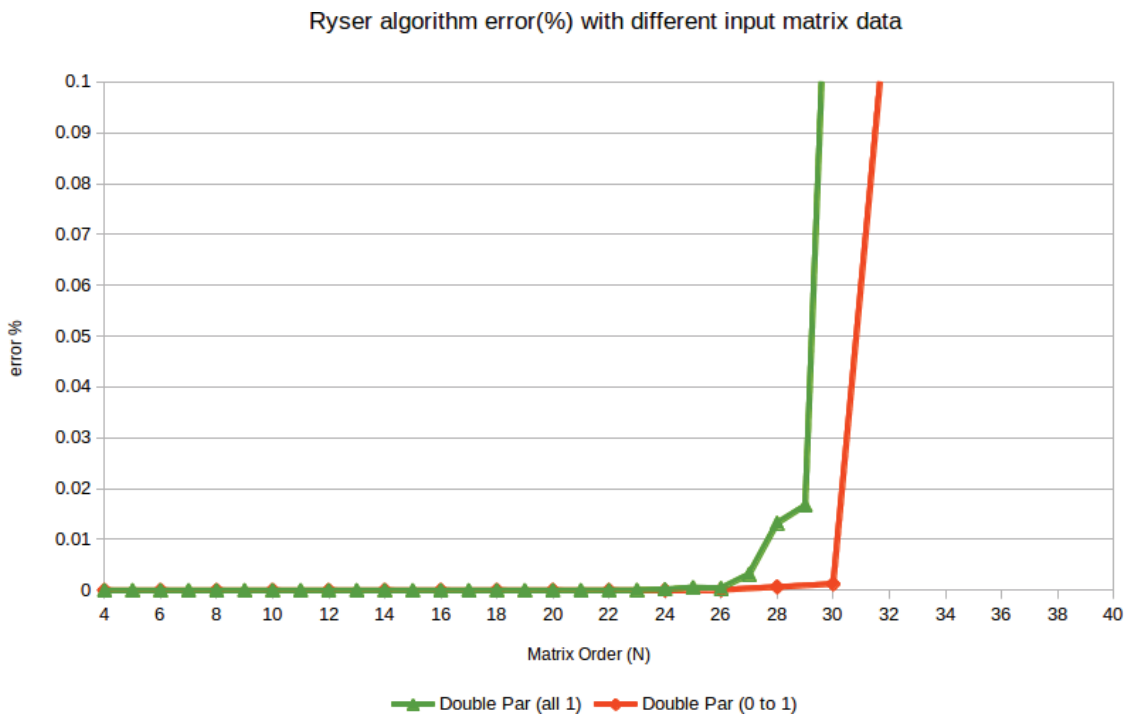

Figure 5.5: Influence of the matrix data values on the parallel Ryser algorithm relative error.

part of the floating point data is so big that there is a loss of precision, not enough significant decimals. The data type that can better manage an increase in matrix size is the *Float80bits* data type in the parallel algorithm. *Float80bits* performs better than *GMP* data type because it has a bigger mantissa (64 bits compared to 50 bits) and more significant decimals (18 compare to 15).

Another performance measurement that has been done is to assess the influence of the input matrix data values on the relative error. Results can be seen in Figure 5.5. These tests have been performed with the *Double* data type and the parallel version of the Ryser algorithm. One test was done with matrices of ones (*Double Par (all 1)*) and the other with matrices filled with random values from zero to one (*Double Par (0 to 1)*). The error exponential rise arrives with at a lower matrix size for matrices of ones. It is explained by the fact that the permanent value is larger with all matrix elements set to one and then the proportion of the significant decimal is lower than total decimal.

5.3 Recommendations

5.3.1 Ryser algorithm

The parallel Ryser implementation is promising because we gained speed compared to the sequential implementation. The best speedup obtained is of 4.5 and is reached with data type *Float80bits*.

The simplest way to improve speed of the Ryser algorithm is to try running it on a more powerful

machine than the one used in this study. The executing time will be improved by increasing the number of processes, a faster CPU and possibly larger memory(cache and RAM). Also, the parallel version could be scaled by using clusters (multiple computers) or cloud computing.

5.3.2 MCMC algorithm

Although the MCMC algorithm is time-polynomial, it does require a lot of processing and is memory intensive. However a Monte Carlo approach is a good candidate for parallelization and we definitively recommend to parallelize it once implemented.

We recommend using an existing Monte Carlo C++ template library, such as the vSMC [26]. This C++ Monte Carlo framework can be used to do sequential and parallel version. This particularity is interesting to perform the speedup performance tests. Different versions of parallel algorithms can be quickly implemented such as multi-threads (many processes on a same computer), clusters (many computers) or using a specialized massive parallel hardware such as Graphics Processing Unit (GPU). The vSMC author experimented a speedup of 15 times using GPUs, which make them particularly interesting for parallelization. Another advantage of using this framework is its extensibility: it is possible to improve the library for specific application.

5.3.3 Importance sampling

As mentioned in section 2.2.2.1, we believe that further research to develop an importance sampling approach would worth a try. Matrices involved in our context have duplicate rows and/or columns. So it may be possible to develop an IS algorithm that uses that characteristic to produce good estimates. This recommendation requires a strong statistical background because it implies to understand how to bias the sampling probabilities to result in an efficient IS estimator.

5.3.4 Concurrency

Another improvement could be to execute on distributed computers different versions of parallel algorithms and use the output of the quickest implementation. This solution involves using a master controller that starts algorithms on the different computers and wait for the first to finish. The idea behind this suggestion is that algorithm performance is usually influenced by the matrix characteristics (e.g. size, sparsity, etc.). This *race* would avoid performing pre-processing on the matrices to decide which algorithm is the best suited.

This page is intentionally left blank.

Bibliography

[1] Gray code, 2014.

[2] Alexander Barvinok. Polynomial time algorithms to approximate permanents and mixed discriminants within a simply exponential factor. *Ann Arbor*, 1001(48109):1109, 1999.

[3] Isabel Beichl and Francis Sullivan. Approximating the permanent via importance sampling with application to the dimer covering problem. *J. Comput. Phys.*, 149(1):128–147, February 1999.

[4] Ivona Bezáková, Daniel Stefankovič, Vijay V. Vazirani, and Eric Vigoda. Approximating the permanent in o(n7) time, 2004.

[5] Ivona Bezáková, Daniel Stefankovič, Vijay V. Vazirani, and Eric Vigoda. Accelerating simulated annealing for the permanent and combinatorial counting problems. *SIAM J. Comput.*, 37(5):1429–1454, 2008.

[6] Andrei Z. Broder. How hard is to marry at random? (on the approximation of the permanent). In *STOC*, pages 50–58, 1986.

[7] Yuguo Chen and Jun S Liu. Sequential monte carlo methods for permutation tests on truncated data. *Statistica Sinica*, 17(3):857, 2007.

[8] Michael Chertkov and Adam B. Yedidia. Approximating the permanent with fractional belief propagation. *J. Mach. Learn. Res.*, 14(1):2029–2066, January 2013.

[9] David G. Glynn. The permanent of a square matrix. *Eur. J. Comb.*, 31(7):1887–1891, 2010.

[10] Bert C. Huang and Tony Jebara. Approximating the permanent with belief propagation. *CoRR*, abs/0908.1769, 2009.

[11] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 712–721, New York, NY, USA, 2001. ACM.

[12] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.

[13] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

[14] Narendra Karmarkar, R Karp, R Lipton, László Lovász, and Michael Luby. A monte-carlo algorithm for estimating the permanent. *SIAM Journal on Computing*, 22(2):284–293, 1993.

[15] P. W. Kasteleyn. The statistics of dimers on a lattice : I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27:1209–1225, December 1961.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009.

[17] SC Kou and P McCullagh. Approximating the α-permanent. *Biometrika*, 96(3):635–644, 2009.

[18] N.Yu. Kuznetsov. Computing the permanent by importance sampling method. *Cybernetics and Systems Analysis*, 32(6):749–755, 1996.

[19] Mark R. Morelande. Joint data association using importance sampling. In *FUSION*, pages 292–299, 2009.

[20] H.J. Ryser. *Combinatorial mathematics*. Carus mathematical monographs. Mathematical Association of America; distributed by Wiley [New York, 1963.

[21] Dominic Schaub. Identity resolution in wide-area surveillance. *Submitted to Information Fusion*, 2013.

[22] Peter. Smith and Brian. Dawkins. Estimating the permanent by importance sampling from a finite population. *Journal of Statistical Computation and Simulation*, 70(3):197–214, 2001.

[23] Shang-Hua Teng, editor. *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*. SIAM, 2008.

[24] Jeffrey K. Uhlmann. Matrix permanent inequalities for approximating joint assignment matrices in tracking systems. *Journal of the Franklin Institute*, 341(7):569–593, 2004.

[25] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[26] Yan Zhou. vsmc: Parallel sequential monte carlo in c++. *CoRR*, abs/1306.5583, 2013.

Appendix A

Testing machine specifications

CPU
Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz[1]

RAM
32GB DIMM DDR3 Synchronous 1600 MHz (0.6 ns)

Number of Cores
4

Number of Threads available
8 (Intel® Hyper-Threading Technology)

Operating System
Unbuntu Server 12.04

[1] `http://ark.intel.com/products/65523`

This page is intentionally left blank.